1 Prolog

Název jazyka je odvozen ze dvou slov: **PRO**gramování v **LOG**ice. Tím současně říkáme, z jakých principů jazyk vychází. Jeho úspěšné používání v praxi dalo základ nové disciplíně matematické informatiky - logickému programování.

1.1 Histroie

Jazyk Prolog má svůj původ z Francie, kde v roce 1972 na univerzitě v Luminy (Marseille) vznikl. Byl navržen jako prostředek k odvozování logických důsledků formulí predikátové logiky. V roce 1974 byl způsob práce analyzován a vznikl teoretický model na základě procedurální interpretace Hornových klauzulí. Další implementace jazyka v roce 1977 pro počítače DEC-10 určila do jisté míry syntaxi jazyka.

Za významný zlom ve vývoji jazyka Prolog je považován rok 1981, kdy japonský projekt počítačů 5. generace si vybral tento jazyk jako základní pro logický procesor centrální jednotky počítače. Tak se začal Prolog prosazovat vedle jiných jazyků pro symbolické výpočty a společně s jazykem Lisp dnes tvoří dvojici nejpoužívanějších jazyků v oblasti umělé inteligence.

1.2 Základní rysy Prologu

Od počátku byl Prolog využíván při zpracování přirozeného jazyka, pro symbolické výpočty, při projektování, konstrukci překladačů, jako prostředek pro reprezentaci a zpracování znalostí a pro řešení úloh.

Program napsaný v Prologu definuje konečný počet relací pomocí příkazů, které popisují jejich vlastnosti. Dá se říci, že program definuje vše, co můžeme vypočítat a odvodit. Klasický problém vytvoření podrobného plánu pro tuto činnost (rozumějme posloupnost příkazů) na první pohled ustupuje do pozadí. Tím se Prolog podobá specifikačním jazykům, které dovolují oddělit problém specifikace od problému efektivního průběhu vytvářeného programu. Výhoda je zcela zřejmá: je snadné upravit specifikaci, horší je upravovat hotový program.

Máte už programátorskou praxi a přesto jste předchozímu odstavci nerozuměli? Klidně pokračujte dál a věžte, že programování v Prologu je zásadně odlišné od klasických programovacích jazyků. Jazyk je postaven jen na několika srozumitelných a jasných principech, které budou v následujícím textu probrány. Po prvních třech až pěti hodinách praktických zkušeností s Prologem se klidně k předchozímu odstavci vraťte a určitě mu budete rozumět.

1.3 Databáze Prologu

Programování v Prologu spočívá v definici známých faktů o objektech a jejich vzájemných vztazích. Dále pak v definici pravidel o objektech a relacích

platících mezi nimi. A nakonec zodpovídání dotazů.

V logickém programování jsou *fakta* nepodmíněné příkazy a *pravidla* příkazy podmíněné. Vše ukládáme do jedné databáze. Prolog nerozlišije mezi daty a programem.

Jednotlivé možnosti Prologu si probereme na příkladu. V literatuře se nečastěji používá pro názornost a jednoduchost příklad rodinných a příbuzenských vztahů. Každý je schopen si příklad přizpůsobit své situaci a ověřit pravdivost zavedených pravidel. Proto se budeme tohoto příkladu držet i zde.

1.3.1 Fakta

Fakta by měla mít určitou vypovídací schopnost. Jen strohé konstatování faktu, i když je pravdivý, nemá smysl. Např.

```
muz.
zena.
```

Vypovídací schopnost takového faktu je pro další použití v programu prakticky nulová. Chceme-li tedy definovat databázi vhodnou k dalšimu použití, musíme zavést kromě objektů i jejich vlastnosti, vztahy, nebo chování.

```
muz( pavel ).
muz( jiri ).
muz( martin ).
muz( frantisek ).

zena( jana ).
zena( petra ).
zena( libuse ).
zena( irena ).
```

Takto už víme, že Pavel je muž, Jana je žena atd.

Všimněte si, že veškerá fakta píšeme malými písmeny. Velkými písmeny začínají názvy proměnných (o nich až za chvíli). Na to je třeba si na počátku dávat pozor!

Také si všimněte, že stejná fakta je třeba uvádět pohromadě s ostatními souvisejícími. Nelze zapisovat střídavě muže a ženy. A na konci každého faktu je tečka.

Fakta nemusí být jen v jednoduchém tvaru, kdy se definuje vlastnost či chování objektu. Můžeme zavádět i vztahy mezi objekty:

```
rodic( frantisek, jana ).
rodic( libuse, jana ).
rodic( frantisek, petra ).
rocic( irena, petra ).
```

```
manzele( frantisek, libuse ).
manzele( jana, jiri ).
manzele( martin, petra ).
```

Takto zavedeme vztah, kterým říkáme, že František a Libuše jsou manželé a jsou rodiči Jany. Všimněte si, že fakt *rodič* říká, kdo je rodičem koho. Zatímco fakt *manželé* už nevyžaduje dodržení pořadí argumentů.

Proto si při zadávání faktů dávejte pozor. Když se rozhodnete pro určité pořadí parametrů, musíte jej dodržovat!

Informaci o manželích je třeba brát jako aktuální stav. Nezachycujeme dřívější vztahy.

Počet parametrů faktů není nijak omezen, je třeba si jen dávat pozor, zda jsou ukládané informace srozumitelné a nepopírají se navzájem.

Počet parametrů faktu nazýváme v Prologu *aritou*.

1.3.2 Dotazy

Cesta, jak se k informacím uloženým v databázi prologu dostat, jsou dotazy. Zadáváme je přímo na příkazovém řádku interpretu Prologu:

```
?- muz(jiri).

Yes
?- muz(pepa).

No
?- manzele(jiri,jana).

No
?- manzele(jana,jiri).

Yes
?-
```

Takto kladené otázky ovšem nejsou příliš praktické a přínosné. Aby byl systém dotazů použitelný v praxi, musíme zavést další dosud chybějící prvek Prologu. Proměnné.

1.3.3 Dotazy během načítání databáze interpretem

Často programátor potřebuje provést určité operace již během načítání dat ze souboru. K tomu slouží operátor '**:-**', který se uvede před jednotlivé dotazy na začátek řádku, např:

```
:-dynamic( prvek/1 ).
:-op( 200, xfx, rodic ).
```

1.3.4 Proměnné

V Prologu není nutno proměnné předem deklarovat, nebo určovat jejich typ. Proměnné mohou být jen ve dvou stavech: *volné* (*nenavázané*), nebo *vázané* na konkrétní atom. Volné proměnné se mohou ztotožnit s libovolným atomem a tím dojde k jejich navázání. Navázané proměnné se mohou ztotožňovat pouze se stejným faktem, na jaký už jsou navázány. K uvolnění proměnné může dojít jen u toho faktu, kde se proměnná navázala, a to při hledání dalšího řešení, nebo při nenalezení dalšího řešení, když dojde k selhání.

Existence proměnné je jen po dobu trvání dotazu. Z předchozího textu také vyplývá, že databáze prologu obsahuje jen fakta a pravidla, nikdy ne proměnné! Počáteční obvyklá chyba je, pokoušet se proměnné ukládat do databáze.

Chceme-li použít proměnnou, stačí uvést identifikátor začínající velkým písmenem.

```
?- muz(Kdo).

Kdo = pavel
```

V této chvíli se proměnná *Kdo* navázala na atom *pavel* a interpret tuto informaci vypsal. Na uživateli je rozhodnutí, zda chce hledat ještě další možná řešení - může odpovědět stiskem klávesy '**;**', nebo zda je s daným zjištěním spokojen a stiskem klávesy '**Enter**' další hledání ukončí. V prvním případě bude výpis vypada následovně:

```
?- muz(Kdo).

Kdo = pavel ;

Kdo = jiri ;

Kdo = martin ;

Kdo = frantisek ;

No
?-
```

Pokud však uživateli stačí pouhé zjištění, že v databázi je alespoň jeden muž a nemá žádné další požadavky, pak může výpis vypadat následovně:

```
?- muz(Kdo).

Kdo = pavel

Yes
?-
```

Hledání může uživatel ukončit kdykoliv uzná za vhodné, není třeba hledat všechna řešení.

Proměnnou můžeme také použít k hledání rodičů Jany:

```
?- rodic(R, jana).

R = frantisek ;

R = libuse ;

No
?-
```

A nejsme omezeni na použití jedné proměnné. Lze v dotazu použít i více proměnných a ptát se na osoby, které jsou manželé:

```
?- manzele(M1,M2).

M1 = frantisek
M2 = libuse ;

M1 = jana
M2 = jiri ;

M1 = martin
M2 = petra ;

No
?-
```

Takto získáme všechny aktuální manželské dvojice.

1.3.5 Pravidla

Ze známých faktů uvedených v databázi jsme schopni odvodit řadu dalších informací. Pro taková odvození používáme dotazy složené pomocí logického součinu (dotazy oddělujeme čárkou). Například víme, že nalézt dva sourozence znamená najít takové osoby, které mají společného rodiče. A musíme vyloučit skutečnost, že každý je svým vlastním sourozencem:

```
?- rodic(R,D1),rodic(R,D2),D1\=D2.

R = frantisek
D1 = jana
D2 = petra ;

R = frantisek
D1 = petra
D2 = jana ;

No
?-
```

Uvedený dotaz můžeme považovat za obecnou definici pro sourozence a z takového dotazu snadno vytvoříme *pravidlo*:

```
sourozenec( D1, D2 ) :-
    rodic( R, D1 ),
    rodic( R, D2 ),
    D1 \= D2.
```

Pokud víme, že existuje více alternativ vedoucích k odvození, zapisujeme příslušná pravidla pod sebe:

```
manzelka( M ) :-
    zena( M ),
    manzele( M, _ ).
manzelka( M ) :-
    zena( M ),
    manzele( _, M ).
```

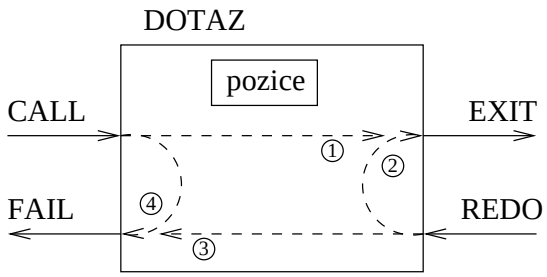
V pravidle pro nalezení nebo ověření manželky jsme použili na místě proměnné, jejíž hodnota pro nás není důležitá, podtržítko. Říkáme, že jde o *anonymní* proměnnou.

Již definovaná pravidla můžeme dále používat pro odvození dalších vztahů. Například pro nalezení sester můžeme definovat následující pravidlo:

```
sestry( S1, S2 ) :-
    sourozenec( S1, S2 ),
    zena( S1 ),
    zena( S2 ).
```

1.4 Provádění programu

Princip činnosti Prologu je založen na tzv. *zpětném řetězení*. Jde v podstatě o jednoduchý princip. Už u dotazů jste si mohli všimnout, že každý dotaz



Obrázek 1: Blokové schéma dotazu v Prologu

začíná prohledávat celou databázi od začátku a při každém splnění si pamatuje, kde prohledávání skončilo. Od zapamatované pozice pak pokračuje v prohledávání dál, pokud je o to požádán. A pokud už žádné další řešení neexistuje, dotaz končí neúspěchem.

Zpětné řetězení se projeví v okamžiku, kdy dotaz bude složen jako logický součin několika dotazů. Pro jeho splnění je nutné, aby byly splněny všechny dotazy. Pokud se některý dotaz splnit nepodaří, vrátí se interpret Prologu k dotazu předcházejícímu a hledá jeho další pravdivé řešení.

Řešení se tedy hledá ve složeném dotazu zleva doprava a při neúspěchu se vracíme k novému plnění zprava doleva. Celý proces zpětného řetězení končí v okamžiku, kdy už první nejlevější dotaz nemá další plnění.

Pro lepší představu si můžeme princip činnosti Prologu znázornit graficky.

1.4.1 Blokový model

Každy dotaz v Prologu můžeme znázornit jako blok se dvěma vstupy a dvěma výstupy podle obrázku 1:

CALL je prvním vstupem při volání dotazu,

REDO slouží jako vstup při požadavku o opětovné plnění,

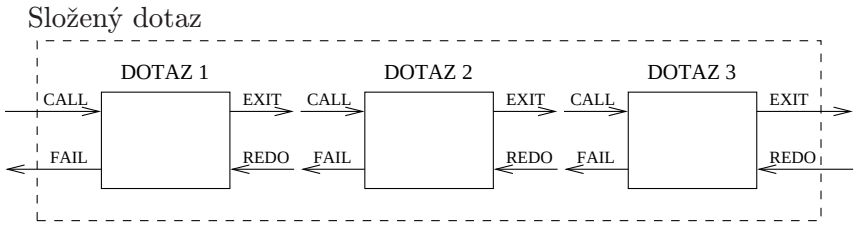
EXIT je výstup při úspěšném splnění dotazu,

FAIL výstupem končí neúspěšné plnění dotazu.

Blok má vnitřní paměť, kam se ukládá pozice postupného prohledávání databáze. Každým vstupem *CALL* se tato pozice nastavuje na začátek databáze.

Vnitřkem bloku vedou čtyři možné průchody:

1. první dotaz byl úspěšný,
2. požadavek na nové plnění byl úspěšný,



Obrázek 2: Blokové schéma složeného dotazu

3. neexistuje další plnění,

4. neexistuje ani jediné řešení dotazu.

Tento blokový model je velmi vhodný i v případě, že chceme znázornit složený dotaz. Jak je vidět z obrázku 2, dotazy z logickém součinu překreslené do blokového schématu na sebe plynule navazují.

Vazba *FAIL-REDO* znázorňuje princip zpětného řetězení.

Co je ovšem také dobře vidět na obrázku 2, to je chování složeného dotazu. Ten se chová navenek jako jediný blok, v obrázku označený čárkovaným obdélníkem. Tímto způsobem lze dotazy do sebe navzájem libovolně skládat a vytvářet prakticky neomezenou hloubku vnoření. Dotaz ve svém důsledku může obsahovat i sám sebe a vytvářet tak rekurzi.

1.5 Řízení a sledování výpočtu

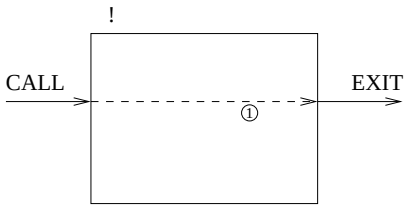
Prolog disponuje prostředky, jak ovlivňovat průběh programu. Princip zpětného řetězení lze ovlivňovat čtyřmi základními způsoby:

- řez,
- repeat,
- true,
- fail.

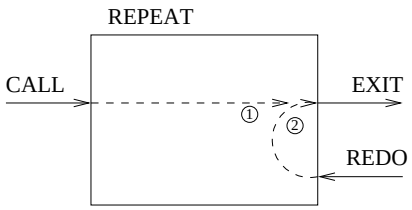
Chování jednotlivých způsobů si postupně vysvětlíme s použitím blokového modelu.

1.5.1 Řez

Jde o zásadní způsob změny řízení zpětného řetězení. Řez se v programu Prologu realizuje jako '**!**' (vykřičník). Podle blokového modelu na obrázku 1 jde o speciální blok, který nemá vstup *REDO* a výstup *FAIL*. Má pouze jediný vnitřní průchod označený jako *1*.



Obrázek 3: Blokové schéma řezu **!**



Obrázek 4: Blokové schéma predikátu **repeat**

Zařazením řezu mezi dotazy se znemožní zpětné řetezení a tím lze ovlivnit postup plnění dotazů.

Jak vypadá výsledný blokový model je na obrázku 3.

1.5.2 Repeat

Toto pravidlo *repeat* slouží pro vytváření cyklů. Je definováno následovně:

```
repeat.
repeat :- repeat.
```

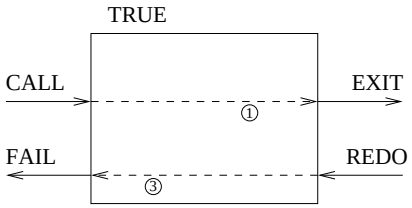
Výsledné chování podle blokového modelu na obrázku 1 je takové, že chybí výstup *FAIL* a zůstávají jen vnitřní průchody *1 a 2*. Dostaneme tak blok podle obrázku 4.

1.5.3 True

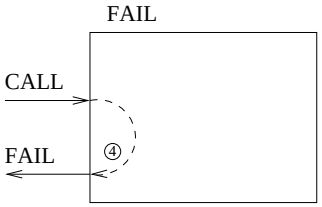
Pravidlo *true* je vždy splněno a při zpětném řetězení se znovu neplní. Na obrázku 1 můžeme v blokovém modelu vypustit dva vnitřní průchody *2 a 4*. Schémat pak bude vypadat podle obrázku 5.

1.5.4 Fail

Poslední pravidlo *fail* není nikdy splněno. Na obrázku 1 můžeme v blokovém modelu nechat jen vstup *CALL*, výstup *FAIL* a vnitřní průchod *4*. Funkcionalita tak bude odpovídat bloku na obrázku 6.



Obrázek 5: Blokové schéma predikátu **true**



Obrázek 6: Blokové schéma predikátu **fail**

1.5.5 Sledování běhu programu

Nejen na začátku práce s Prologem se asi každý dostane do situace, kdy má dojem, že jeho program dělá něco jiného, než autor zamýšlel. Pro ladící účely jsou v Prologu následující predikáty:

```
spy( predikát ).
trace.
notrace.
```

Predikátem `spy` se definují ty dotazy, které chceme sledovat. Použijeme-li výraz z trasovacího nástroje pro klasický programovací jazyk, jde o bod zastavení.

Predikáty `trace` a `notrace` pak samotný systém sledování postupu dotazování Prologu zapnou nebo vypnou.

1.6 Práce s databází

Prolog disponuje mechanismem, jak dynamicky za chodu měnit obsah databáze. Není ale nutno omezovat práci jen na fakta, je možno přidávat a odebírat i pravidla. Slouží k tomu čtyři předdefinované predikáty:

```
assert( Fakt ), assertz( Fakt )
```

Přidání faktu na konec databáze.

```
asserta( Fakt )
```

Přidání požadovaného faktu na začátek databáze.

```
retract( Fakt )
```

Odebere z databáze první výskyt požadovaného faktu.

1.6.1 Fakta - dynamic

Interpret Prologu obsahuje určité kontrolní mechanismy, jak za běhu programu upozornit programátora, že se s databází pracuje neočekávaným způsobem.

Pokud víte, že budete potřebovat za běhu programu odebírat určitá fakta, musíte pomocí predikátu `dynamic( fakt/arita )` tuto skutečnost předem sdělit. Predikát `dynamic` musíte také použít, pokud budete chtít do databáze přidávat fakta a to zejména pokud v databázi ještě žádný požadovaný fakt daného jména neexistuje.

Je dobrým zvykem uvádět pro lepší přehlednost všechny dynamické predikáty hned na začátku zdrojového textu, např:

```
:-dynamic( cesta/2 ).
:-dynamic( mesto/1 ).
:-dynamic( prvek_sezn/1 ).

mesto( ostrava ).
mesto( praha ).
...
```

1.7 Matematické výrazy

Veškeré výrazy jsou v Prologu uchovávány v symbolické podobě. Ve výrazech můžeme používat čísla i vázané proměnné a obvyklé matematické operace `+ - * / mod`, kdy interpret dodržuje priority operací. Pokud ale chceme symbolický zápis vyhodnotit, musíme použít speciální příkaz **is**.

Následující príkazy asi nepotřebují komentář:

```
X is 2 * 3 + 1
N is A * 3
O is 3.14 * R
Z is X + Y
N is M + 1
```

Pozor je třeba si dát v případech, kdy tvoříme např. čítače. Již navázanou proměnnou nelze znovu navázat. Proto je následující příkaz chybný a je třeba použít některý z již dříve uvedených příkazů:

```
N is N + 1    % chybný příklad
```

1.8 Předdefinované predikáty

Pro začátek práce s Prologem je třeba znát několik základních vestavěných predikátů. Podrobnější informace najdete v dokumentaci SWI-Prologu.

```
consult( soubor ).
[soubor].
```

Natažení databáze ze souboru do interpretu. Předpokládá se, že soubor je v aktuálním adresáři, jeho jméno neobsahuje velká písmena a má příponu `.pl`.

```
halt.
```

Příkazem `halt` ukončíte práci s interpretem Prologu. Asi se budete zpočátku intuitivně pokoušet ukončit práci pomocí příkazů `exit`, `quit` nebo `bye`, ale v tomto je Prolog neodbytný. Jedině `halt`.

```
listing.
listing( fakt ).
listing( fakt/arita ).
```

Výpis obsahu databáze. Pokud použijeme `listing`, vypíše se celá databáze. Uvedením názvu faktu však můžeme celý výpis omezit jen na požadovanou část.

```
write( X ).
display( X ).
nl.
```

Predikáty pro výpis informací. Argumentem je vždy jen jeden fakt. Predikát `display` vypisuje fakt ve vnitřním prefixovém formátu, zatímco predikát `write` provádí výpis v infixové, tedy uživateli přijatelnější podobě. Poslední uvedený predikát `nl` slouží pro přechod kurzoru na nový řádek a nemá

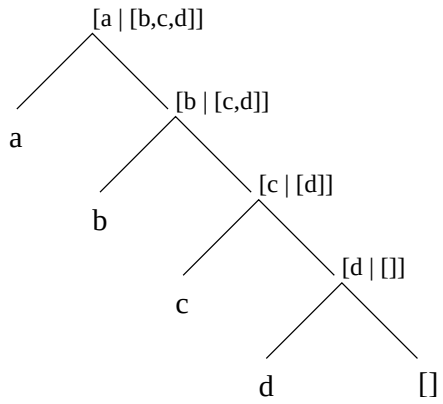
Vyzkoušejte si jednoduchý příklad `write( 1+2+3 )` a `display( 1+2+3 )`.

```
read( X ).
```

Načtení faktu, nebo spíše jen atomu, z klávesnice. Nezapomeňte, že každá zadávaná informace musí být ukončena tečkou!

```
help( predikát ).
```

Vypíše nápovědu k uvedenému predikátu, pokud je znám.



Obrázek 7: Vnitřní struktura seznamu [a,b,c,d]

2 Seznamy

Prolog disponuje možností, jak pracovat s více fakty současně a sdružovat je do jedné struktury - *seznamu*. Seznamy jsou dynamické struktury definované rekurzivně. Jak je vidět na obrázku 7, každý seznam se vždy skládá z *hlavy* a *těla* (*head* and *tail*). Za posledním prvkem následuje prázdný seznam, aby byla splněna rekurzívní definice.

V zápise seznamu používáme čárku pro oddělení prvků seznamu a znak svislice (pipe) pro oddělení hlavy a těla seznamu. Uveďme několik příkladů.

Čtyřprvkový seznam:

```
[ a, b, c, d ]
```

Seznam rozdělený na hlavu a tělo (musí mít munimálně jeden prvek):

```
[ H | T ]
```

Prázdný seznam:

```
[]
```

Seznam začínající dvěma konkrétními prvky:

```
[ x, y | T ]
```

Dvouprvkový seznam, kde druhým prvkem je seznam:

```
[ a , [ 1, 2 ] ]
```

Předchozí přiklad nezaměňovat chybně s tříprvkovým seznamem:

```
[ a | [ 1, 2 ] ]
```

2.1 Práce se seznamy

Pro rekurzivně definované seznamy je většina algoritmů také rekurzívních. Ukážeme si jen několik základních operací se seznamy, další už zůstávají na čtenáři.

Nejzákladnější operace je zjištění délky seznamu:

```
% delka( I, O ) - Seznam, Délka
delka( [], 0 ).
delka( [ H | T ], N ) :-
    delka( T, N1 ),
    N is N1 + 1.
```

U všech rekurzívních algoritmů je třeba vždy stanovit správně limitní podmínku. Na to je třeba myslet hned na začátku. A pak následuje realizace rekurze.

Další operací je přidání prvku na začátek seznamu. To je velmi jednoduché:

```
% pridej_hlavu( I, I, O ) - Seznam, Prvek, Výstupní_seznam
pridej_hlavu( S, P, [ P | S ] ).
```

Přidat ale prvek na konec seznamu je složitější, protože ke konci se nejprve musíme dopracovat, vložit tam požadovaný prvek a vrátit všechny prvky zpět:

```
% pridej_konec( I, I, O ) - Seznam, Prvek, Výstupní_seznam
pridej_konec( [], P, [ P ] ).
pridej_konce( [ H | T ], P, [ H | O ] ) :-
    pridej_konec( T, P, O ).
```

A mezi základní operace patří ověření, zda je konkrétní prvek obsažený v seznamu:

```
% prvek( I, I ) - Prvek, Seznam
prvek( [ P | T ],  P ).
prvek( [ H | T ], P ) :-
    prvek( T, P ).
```

3 Operátory

Veškerá data jsou v Prologu vnitřně ukládána v prefixové notaci. Pro uživatele je ovšem pohodlnější a hlavně přehlednější používání infixové notace. Pokud ovšem připustíme používání operátorů, musíme definovat několik vlastností: priorita, asociativita a pozice operátoru.

Priorita je číslo v intervalu $< 1, 1200 >$, kde vyšší číslo znamená nižší prioritu.

Asociativita může a nemusí být povolena a v případě infixového operátoru se musí určovat, zda je asociativita zleva, nebo zprava.

Poslední co nás zajímá, je pozice operátoru. Ta může být prefixová, infixová a postfixová.

Pro definici asociativity a pozice se používají následující symboly:

```
xfx xfy yfx     % infix
fx fy           % prefix
xf yf           % postfix
```

kde značka **f** je pozice operátoru, symbol **x** nedovoluje asociativitu a **y** naopak asociativitu povoluje.

3.1 Předdefinované operátory

Pokud vás zajímá, jaké operátory jsou definované a jakou mají prioritu, můžete zkusit následující dotaz (výpis byl pro přehlednost setříděn podle priority):

```
?- current_op( X, Y, Z ), write( [ X, Y, Z ], nl, fail.
[1, fx, $]
[200, xfx, **]
[200, xfy, ^]
[400, yfx, *]
[400, yfx, //]
[400, yfx, /]
[400, yfx, <<]
[400, yfx, >>]
[400, yfx, mod]
[400, yfx, rem]
[400, yfx, xor]
[500, fx, +]
[500, fx, -]
[500, fx, ?]
[500, fx, \]
[500, yfx, +]
[500, yfx, -]
[500, yfx, /\]
[500, yfx, \/]
[600, xfy, :]
[700, xfx, <]
[700, xfx, =..]
[700, xfx, =:=]
[700, xfx, =<]
[700, xfx, ==]
```

```
  [700, xfx, =@=]
  [700, xfx, =\=]
  [700, xfx, =]
  [700, xfx, >=]
  [700, xfx, >]
  [700, xfx, @<]
  [700, xfx, @=<]
  [700, xfx, @>=]
  [700, xfx, @>]
  [700, xfx, \==]
  [700, xfx, \=@=]
  [700, xfx, \=]
  [700, xfx, is]
  [900, fy, \+]
  [1000, xfy, (,)]
  [1050, xfy, (*->)]
  [1050, xfy, (->)]
  [1100, xfy, (;)]
  [1100, xfy, (|)]
  [1150, fx, (discontiguous)]
  [1150, fx, (dynamic)]
  [1150, fx, (initialization)]
  [1150, fx, (meta_predicate)]
  [1150, fx, (module_transparent)]
  [1150, fx, (multifile)]
  [1150, fx, (volatile)]
  [1200, fx, (:-)]
  [1200, fx, (?-)]
  [1200, xfx, (-->)]
  [1200, xfx, (:-)]

No
?-
```

3.2 Symbolická matematika

Jak již bylo dříve zmíněno, všechny informace si vnitřně udržuje Prolog ve formě symbolů v prefixové notaci. Tímto získává programátor nástroj s vyjímečnými možnostmi. Práce s matematickými výrazy lze provádět čistě symbolicky, nikoliv jen číselně. Je možno takto například sestavit programy pro derivování matematických výrazů, řešení některých typů matematických úloh, nebo zjednodušování matematických výrazů.

Tato problematika zde ovšem podrobněji probírána nebude.

3.3 Definice vlastních operátorů

Kromě již definovaných operátorů si může každý programátor definovat své vlastní operátory. Slouží k tomu predikát:

```
op( priorita, typ, operátor ).
```

Pokud chcete použít operátory ve svých programech, mějte na paměti, že interpret musí znát operátor dřív, než začne načítat zdrojové programy, kde jsou operátory použity!

Příklad, jak takové použití může vypadat v praxi je následující přepis naší databáze z počátku tohoto textu:

```
:-op( 50, fx, muz ).
:-op( 50, fx, zena ).
:-op( 50, xfx, rodic ).
:-op( 50, xfx, manzele ).

muz pavel.
muz jiri.
muz martin.
muz frantisek.

zena jana.
zena petra.
zena libuse.
zena irena.

frantisek rodic  jana.
libuse rodic jana.
frantisek rodic petra.
irena rodic petra.

frantisek manzele libuse.
jana manzele jiri.
martin manzele petra.
```

4 Pomocné predikáty

Pro řadu implementací algoritmů inteligentních systémů budeme potřebovat ještě několik dalších pomocných predikátů. Stačí se zmínit např. o potřebě evidence nalezených řešení. Okamžitě se samozřejmě nabízí použití predikátu *assert()*. Má to ale jeden háček. Pokud řešení problému selže, algoritmus se obvykle pokusí pomocí zpětného řetězení o hledání dalších řešení. V databázi

ale již přidané informace zůstanou, protože *assert()* neprovádí při zpětném řetězení žádné akce. Musíme tedy najít způsob, jak informace do databáze přidávat, ale při zpětném řetězení je opět odstranit.

V řadě případů potřebujeme také zabránit, abychom do databáze nepřidali informaci, která už v ní uložena je. Důvody mohou být různé, např. omezený počet zdrojů, nebo se při řešení nechceme vracet na místa, kde jsme už byli.

A celý problém může být i opačný, kdy algoritmus data z databáze odebírá, tedy spotřebovává. Při selhání algoritmu však je nutno odebrané informace či prostředky do databáze vrátit zpět.

Podívejme se tedy na několik základních variant pomocných predikátů pro práci s databází.

4.1 Přidání s následným úklidem - assertu

Název *assertu* je mnemotechnicky odvozen od assert „s **Ú**klidem“.

Cílem tohoto predikátu je přidat data do databáze a při backtrackingu je opět ostranit. Opětovné plnění predikátu musí ovšem selhat, i když je dle našich představ splněné!

```
assertu( Fakt ) :-
                      assert( Fakt ).
assertu( Fakt ) :-
        retract( Fakt ), !, fail.
```

4.2 Přidávání bez opakování - assertnd

Název *assertnd* je mnemotechnicky odvozen od assert „**N**e **D**uplicity“.

Úkolem tohoto predikátu bude přidávat fakta do databáze, při zpětném řetězení je odebrat a nepovolit přidání faktů již v databázi obsažených, tedy bez duplicit.

Predikát bude vypadat následovně:

```
assertnd( Fakt ) :-
          Fakt, !.  % fakt je databazi - hotovo
assertnd( Fakt ) :-
                      assert( Fakt ).
assertnd( Fakt ) :-
        retract( Fakt ), !, fail.
```

Pozor na řez, tedy vykřičník, na konci první klauzule. Jeho význam je důležitý. Pokud už fakt v databázi obsažen je, tak my jsme právě v této chvíli nebyli ti, kdo ho tam přidal, a tak jej ani při backtrackingu nechceme a ani nesmíme odstranit!

4.3 Přidávání s blokováním - assertb

Predikát má svůj název odvozen od assert „**B**lokující“.

Tento třetí pomocný predikát je nejpoužívanější. Jeho úkol je zdánlivě velmi podobný, jako u *assertnd* - v databázi se nesmí požadovaná informace opakovat. Je zde ale jeden zásadní rozdíl, *assertb* selhává již při zjištění, že informace je v databázi obsažena.

Predikát vypadá takto:

```
assertb( Fakt ) :-
          Fakt, !, fail.   % fakt je v databazi - nelze!
assertb( Fakt ) :-
                      assert( Fakt ).
assertb( Fakt ) :-
        retract( Fakt ), !, fail.
```

4.4 Predikáty pro odebírání - retract*

Názvy predikátů *retract** mají obdobný význam a analogicky opačnou funkcionalitu, jako predikáty *assert**. Popis zde již uvádět nebudeme, jen pro úplnost uvedeme výsledný tvar:

```
retractu( Fakt ) :-
                      retract( Fakt ).
retractu( Fakt ) :-
        assert( Fakt ), !, fail.


retractnd( Fakt ) :-
          not( Fakt ), !.  % fakt neni databazi - hotovo
retractnd( Fakt ) :-
                      retract( Fakt ).
retractnd( Fakt ) :-
        assert( Fakt ), !, fail.


retractb( Fakt ) :-
          not( Fakt ), !, fail.   % fakt neni v databazi - nelze!
retractb( Fakt ) :-
                      retract( Fakt ).
retractb( Fakt ) :-
        assert( Fakt ), !, fail.
```

5 Cvičení

Práci s interpretem jazyka prolog a tvorbě prvních vlastních programů jsou věnována 2 cvičení.

5.1 Rodinné vztahy

Úkolem pro první cvičení je definování a ověřování rodinných vztahů. Každý student si nakreslí obrázek s rodinnými vztahy. Ve vztazích by měly být zachyceny minimálně 3 generace pro minimálně 15 osob. Grafické znázornění vztahů mezi osobami je pro první cvičení velmi důležité! Nepodceňujte jej. Zakreslené vztahy je nutno následně přenést do databáze prologu a vyzkoušet si dotazování v interpretu.

Programově pak definujte vztahy: prarodič, babička, dědeček, vnuk, vnučka, strýc, teta, neteř, synovec, bratr, sestra a pod.

5.2 Práce se seznamy

Práce se seznamy je velmi důležitou součástí efektivní práce s Prologem. Úkolem pro druhé cvičení je vyzkoušet výše uvedené predikáty pro práci se seznamy. Pro úplnost je třeba doimplemetovat odstranění prvního výskytu prvku ze seznamu, odstranění všech výskytů prvku ze seznamu, spojení dvou seznamů a otočení pořadí prvků seznamu.